CITI Technical Report 06-02

Reliable Replication at Low Cost

Jiaying Zhang
jiayingz@eecs.umich.edu

Peter Honeyman
honey@citi.umich.edu

Abstract

The emerging global scientific collaborations demand a scalable, efficient, reliable, and still convenient data access and management scheme. To fulfill these requirements, this paper describes a replicated file system that supports mutable (i.e., read/write) replication with strong consistency guarantees, small performance penalty, high failure resilience, and good scaling properties. The paper further evaluates the system using a real scientific application. The evaluation results show that the presented replication system can significantly improve the application's performance by reducing the first-time access latency to read the input data and by distributing the verification of data access to a nearby server. Furthermore, the penalty of file replication is negligible as long as applications use synchronous writes at a moderate rate.

January, 2006

Center for Information Technology Integration
University of Michigan
535 West William Street
Ann Arbor, MI 48103-4978

Reliable Replication at Low Cost

Jiaying Zhang
jiayingz@umich.edu

Peter Honeyman
honey@citi.umich.edu

1. Introduction

The scientific community has seen an increasing demand for global collaborations, spanning disciplines from high energy physics, to climatology, to genomic. Applications in these fields make intensive use of computational resources far beyond the scope of a single organization, and require access to massive amounts of data. This imposes new challenges in data access, processing, and distribution.

Driven by the needs of scientific collaborations, the emerging Grid infrastructure [8, 9] aims to connect globally distributed resources to a shared virtual computing and storage system, offering a model for solving large-scale computation problems. The sharing in Grid computing is not merely file exchange but rather the direct access to computers , software, data, and other resources, as is required by a range of collaborative scientific problem-solving patterns.

Presently, the primary data access method used on Grid is GridFTP [21]. Engineered with Grid applications in mind, GridFTP has many advantages: automatic negotiation of TCP options to fill the pipe, parallel data transfer, integrated Grid security, and partial transfers that can be resumed. In addition, as an application, GridFTP is easy to install and support across a broad range of platforms.

While simple and easy to implement, as a remote file transfer protocol, GridFTP does not support sophisticated distributed sharing that many Grid applications would require, which impedes the **convenient** use of globally distributed resources for scientific studies.

For example, in a common Grid use case, a scientist wants to run a simulation on high performance computing systems and analyze results on a visualization system. With the Grid technologies available today, the scientist submits the job to a Grid scheduler, such as Condor-G [11]. The Grid scheduler determines where to run the job, prestages the input data to the running machines, monitors the progress of the running job and when the job is complete, transfers the output data to the visualization system through GridFTP. The output data is reconstructed in the visualization site, and the final results are returned to the scientist after reconstruction.

The scenario is attractive as the scientist now is able to use more computing resources to speed up his simulation. However, the whole process is still performed in a batch mode. The scientist cannot view the intermediate results before the entire scheduled job is complete. Since scientific simulations are problem-prone, needing to wait for hours or days to discover a mistake in the experiment is inefficient.

To facilitate Grid computing over wide area networks, we develop a replicated file system that provides users high performance data access with the standard file system semantics. The system supports a global name space and location independent naming, so applications on any client can access a file with the same name and without needing to know where the data physically locates. It supports mutable replication, i.e., read/write replication, with consistency guarantees, so users can perform data modification easily, safely, and efficiently. The semantics that the system provides is compatible with POSIX API, allowing easy deployment of unmodified scientific applications. We have implemented our design in NFSv4, the emerging distributed file system standard [19]. In latter discussion, we refer to the implemented replication system as **rNFS** for short.

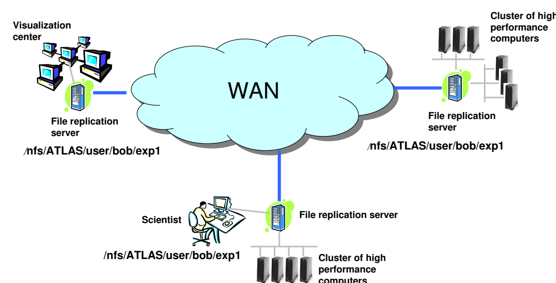

Figure 1: A Grid use case example.
The figure shows a common Grid scenario. In the example, by employing a replicated file system, the remote computation nodes can access data from a nearby server. Meanwhile, the scientist can view intermediate results and adjust experiment in real time.

Under rNFS, the scientist in the example described above can now monitor and control the progress of the simulation in real time. As illustrated in Figure 1, with the support of a global name space, the scientist can run programs on remote machines with the same pathname and without any reconfiguration. By using a replicated file system, the intermediate output of simulation is automatically distributed

to the visualization center and the scientist's computer. The scientist can view intermediate results and determine if parameters or algorithms need to be adjusted. If so, he can update them from his local computer and restart the simulation on the remote site, as simple as if he was running the experiment locally. Meanwhile, the remote computation nodes can still access data from a nearby server.

The remainder of the paper is organized as follows. We first give an overview of the system architecture in Section 2. Then we focus the major part of the paper on application evaluation, with which we examine how the system performs over wide-area networks and what performance benefit it can provide to Grid applications. Section 3 presents the experimental results we collected, as well as the detailed data analysis. Following that, we review the related work in Section 4, and conclude in Section 5.

2. Design and Architecture

This section describes a replicated file system designed to facilitate the data access and management of large-scale scientific applications. Section 2.1 presents the design of a naming scheme that supports a global name space and location independent naming. Following that, Section 2.2 describes a replication protocol that provides mutable replication support with strong consistency guarantees.

2.1. Name Space Design

The NFSv4 protocol includes features to support read-only file system replication using a special file attribute `FS_LOCATIONS`. By the NFSv4 specification, a client's first access to a replicated file system yields the `FS_LOCATIONS` attribute that lists alternative locations for the file system. Complying with the published NFSv4 protocol, we also use the `FS_LOCATIONS` attribute to communicate replica location information between servers and clients. However, the namespace of rNFS includes two extended features.

First, we extend NFSv4 client side to support a global name space that hides server location details from users. By convention, a special directory `/nfs` is the global root of all NFSv4 file systems. Entries under `/nfs` are mounted on demand. The first time a user accesses any NFSv4 file system, the referenced name is forwarded to a daemon that queries DNS to map the given name to one or more file server locations, selects a file server, and mounts it at the point of reference. The format of reference names under `/nfs` follows Domain Name System [16] conventions. We use SRV Resource Record [17] to store server location information. The content of a SRV RR maps a reference name to a list of file servers that hold the copies of data.

The second extended feature is the support for directory replication. We implement directory replication by exporting a directory with an attached reference string that includes information on how to get directory replica locations. When a client first accesses a replicated directory, the server uses the attached reference string to resolve the replica locations of that directory, and sends this information to the client through the `FS_LOCATIONS` attribute.

2.2. Mutable Replication

To meet availability, performance, and scalability requirements, distributed services naturally turn to replication; file service is no exception. While the concept of file system replication is not new, existing solutions either forsake read/write replication totally [4, 23, 26] or weaken consistency guarantees [14, 24]. They fail to satisfy the requirements for global scientific collaborations.

Returning to the example described in Section 1, experiment analysis is usually an iterative, collaborative process. The stepwise refinement of analysis algorithms requires using multiple clusters to reduce development time. Although the workloads during this process are often dominated by read, they also demand the underling system to support write operations. Furthermore, strong consistency guarantees are often taken for granted, e.g., an executable may incorporate user code that is finished only seconds before the submission of the command that requires to use the code.

In this section, we present a mutable file replication protocol that balances the tradeoff among consistency, performance, and failure resilience by offering applications stringent yet flexible consistency guarantees. The protocol can guarantee either ordered writes or synchronized access, without adding overhead on normal reads. It can tolerate a large class of server crash or link failures, even when these lead to network partitioning. Our design uses standard POSIX features, which makes it easy to deploy.

Below, we first describe a replication scheme that guarantees ordered writes. Based on that, we present the additional mechanisms to enforce synchronized access. In the following discussion, we refer the first consistency model as sequential consistency, and the second as synchronized accessed. We note that we only highlights the important features of rNFS here for evaluation purpose. For more design details, readers can refer to another paper [27].

2.2.1. Sequential Consistency

By default, our replication protocol guarantees ordered writes in which replication servers do not necessarily see updates simultaneously, but they are guaranteed to see them in the same order. In this model, when a client opens a file for writing, the chosen server temporarily becomes the primary server for that file. All other replication servers are instructed to forward client write requests for that file to the primary server. When the file is closed, the primary server withdraws from its leading role by notifying other replication servers to stop forwarding writes. In the follow-

ing discussion, we refer to the first procedure as disabling replication, and the latter as re-enabling replication.

The idea of using primary copy to support data replication or backup is not new [12, 15]. However, compared with the traditional primary copy scheme, our design has the following advantages. First, in rNFS, the overhead to support mutable replication is induced only when there are writes happening. If there are no writes, the system behaves as a read-only replication system, i.e., a client accesses data from a nearby server. Second, a primary server is selected on the granularity of a single file, and thus allows fine-grained load balancing. Third, in rNFS, a primary server is dynamically chosen at the time that a file is write opened. So in most cases (exclusive write cases), a client's write requests are served by a nearby primary server. Furthermore, the solution well suits the Grid computing environment where a replica can be dynamically created and it is hard to decide an optimal primary server for a file beforehand. And fourth, we develop a failure recovery mechanism that conforms with the described primary copy model, as next paragraph presents. We note for emphasis that in rNFS, failure detection and recovery are driven by client accesses, so no heartbeat messages or special group communication services are needed.

To guarantee consistency upon failures, every replication server keeps track of the liveness of other servers. The set of live servers is called the active view. To avoid unnecessary network traffic, we do not use periodic heartbeat to maintain active view. Rather, in our system, active view is refreshed during updates. Basically, during file modification, the primary server removes from its active view any server that fails to respond to its request. The primary server can acknowledge a client write request only if it receives acknowledgments from a majority of replication servers. When the file is closed, the primary server sends its active view to other active servers. A server not in the active view may have stale data, so the active servers refuse any later request that comes from a server not in its active view. A failed replication server can rejoin the active group only after it synchronizes with the up-to-date copy.

By requiring a distributed update to reach a majority of replication servers before replying to a client write request, rNFS can automatically recover from a failure (including primary server failure and partition failure) and continuously serve client requests as long as a majority of the replication servers are in working order. In our system, update distribution is performed in parallel. So the system performance is not affected by occasional message delays or the failure of a minority of the replication servers. Furthermore, the response time for a client write request is determined by the median RTT between the primary server and the replication servers. In the latter discussion, we refer this RTT as *majority RTT* for short.

2.2.2. Synchronized Access

To support synchronized access without imposing overhead on applications that require ordered writes only, we provide synchronization guarantee as an option that can be demanded by applications through POSIX synchronization flags in the open system call interface [3].

By the POSIX specification, if an application opens a file with `O_SYNC` flag set, a subsequent write operation is complete only when the written data and all file attributes relative to the operation, e.g., modification time, is written to the permanent storage; if an application opens a file with both `O_SYNC` and `O_RSYNC` flags set, a read operation is complete only when any pending writes affecting the data to be read is successfully transferred to the requesting process.

Our system takes these flags as the hint that the application is demanding synchronized access. When the primary server receives a synchronous write request (i.e., if `O_SYNC` flag of the file is set or if the file owner requests a fsync) from a client, it must ensure that every replication server has acknowledged its role before returning a reply to the client. By default, a replication server forwards write requests only while its replication is disabled. However, if during this period, a client opens the file with synchronous read requirements (i.e., if `O_SYNC` and `O_RSYNC` flags of the file are set), the replication server forwards the client's read requests to the primary server as well.

With the described mechanism, slight overhead is induced to guarantee synchronized access when applications demand it; longer delay is charged on forwarded operations if concurrent writes occur; If a file is not under modification, any read requests for the file, even those with synchronization requirement, are processed by a nearby server.

2.2.3. Summary and Discussion

There are two primary reasons to maintain consistency among replication servers: first, to guarantee data durability (i.e., no data lost) after recovery of failure; and second, to guarantee correctness during concurrent writes. The second reason involves two cases. The first is to guarantee write ordering with multiple writers, and the second is to guarantee synchronization for simultaneous read and write.

To guarantee data durability, we require a distributed update to reach a majority of replication servers before replying to a client write request, so the system can always find a valid copy (i.e., a copy that reflects all the acknowledged writes) in the majority partition if a failure occurs. In addition to that, we develop two consistency models: sequential consistency and synchronized access consistency. The first consistency model guarantees ordered writes with the primary server as a central point to decide the order of writes. The second consistency model further guarantees that a synchronous read reflects the most recent write by ensuring

that every replication server has noticed that the file is under modification upon synchronous writes, and that a synchronous read request is processed by the primary server.

We can observe that in the synchronized access model, a primary server can not be elected even if a single replication server fails. So compared with the sequential consistency model, synchronized access provides stronger consistency guarantee but less failure resilience. Although our system provides different failure resilience in the support of different consistency requirements, these design choices are based on the same principle, namely, to offer applications a reliable data service.

We notice that for scientific applications, losing computation results can cause expensive cost, and sometimes even correctness problems. E.g., scientific applications usually keep track of their computation progress through log files. Suppose that a failure occurs after an application just completes some computation and records that in a log file. In a system that provides no data durability guarantee, even though the log file indicates that the computation has completed, the results may be lost after recovery of failure. Hence, the user cannot tell where the computation should be restarted.

Based on these considerations, we develop a replication protocol that always guarantees durability of written data acknowledged by the server. Under this prerequisite, the system makes the best effort to mask a failure from applications. However, in case of a non-recoverable failure, we elect to report the failure to the application immediately, instead of masking it, which risks losing the results of a computation or executing incorrect programs.

Another issue introduced in the synchronized access model is that existing programs may not use open synchronization flags to specify their consistency requirements as we expect. Thus modifications are required on the program's open calls to ensure synchronized access. As a makeshift, we can provide synchronization support as a mount option so that the current applications can be deployed without any modification. However, we still recommend the proposed approach for it allows applications to control file sharing behavior more flexibly. Consider the example of an edit-and-run procedure. The program is edited on one client, and then a number of clients are instructed to execute it. Because the execution instruction can be issued immediately after the program editing, the access on the file must be coordinated. In rNFS, correct synchronization behavior can be guaranteed if the editor application issues a fsync system call after completing the editing, and the execution application opens the file with both `O_SYNC` and `O_RSYNC` flags set. On the other hand, another application, e.g., a snapshot tool, can choose to open the file without setting any synchronization flags as sequential consistency is sufficient to guarantee its correctness.

3. Evaluation

After highlighting the important features of our system, in this section, we explore the performance of rNFS with a real scientific application over the simulated wide-area networks. The application we use is from the Atlas simulation software, a cluster-based, data-intensive, distributed program poised for deployment in Grid. For evaluation purpose, we focus on the usage scenarios similar to the one depicted in Figure 1. However, we expect that many of our finds apply to other scenarios as well.

We measured all the experiments presented in this paper with a prototype implemented in Linux 2.6.12 kernel. Servers and clients all run on dual 2.8GHz Intel Pentium4 processors with 1024 KB L2 cache, 1 GB memory, and dual Intel 82547GI Gigabit Ethernet cards onboard. The number of bytes NFS uses for reading (rsize) and writing files (wsize) are set as 32768 bytes. In all experiments, we use NistNet simulator [7] to simulate the network delays. All numbers presented are mean values from three trials of each experiment; standard deviations (not shown) are within five percent of the mean values.

Below, after a brief description of the Atlas software, we describe the experiments with these applications and present the evaluation results.

3.1. Atlas Applications

Atlas is a particle physics project that searches for new discoveries in the high-energy proton collisions [1]. The protons will be accelerated in the Large Hadron Collider accelerator, currently under construction at the European Laboratory for Particle Physics (CERN) near Geneva [2]. The accelerator is expected to start operating in 2007. After that, on the order of a petabyte of raw data will be produced each year and distributed to a multi-tiered collection of decentralized sites for analysis. Atlas is the largest collaborative effort ever attempted in the physical sciences. 1800 physicists from more than 150 universities and laboratories in 34 countries participate in this experiment. With the massive amount of data to be processed and the widely distributed collaborators, Atlas stands to benefit from a scalable and reliable data access and management scheme, which is also what our design targets.

Currently, Atlas is performing large-scale simulation of physics events that will occur within an Atlas detector. These simulation efforts support detector design and the development of real-time event filtering algorithms that are critical for controlling the flood of data when LHC accelerator is running.

The Atlas simulation event data model consists of four stages. The first stage, `Event Generation`, uses a seed to produce pseudo-random events drawn from a statistical distribution deduced from other experiments. The second

stage, `Simulation`, reads the generated events and simulates the passage of particles through the detectors. The third stage, `Digitization`, converts simulated hit events into digital outputs (called digits). The digits are fed to the fourth stage, `Reconstruction`, which performs pattern recognition and track reconstruction algorithms, converting raw digital data into meaningful physics quantities. The four stages have different computational requirements and generate different amounts of output data. For example, when processing 1000 events on a dual 2.4 GHz Pentium4 processors with 1 GB memory, `Event Generation` takes two minutes to finish and generates 20 MB of output; `Simulation` stage takes 33 hours and generates 800 MB; `Digitization` takes 8 hours and generates 1.6 GB; and `Reconstruction` takes 8 hours to finish, generating 8 to 20 MB output data.

In this paper, we skip over discussion of the first two stages: the time spent on `Event Generation` is considerably less than the other three, while `Simulation` is utterly CPU bound, thus Atlas performance is not likely to be sensitive to our work on the I/O side. Our analysis focuses on `Digitization` and `Reconstruction`, where we want to investigate the performance of rNFS from the following two aspects.

First, to demonstrate the benefit of server replication for global-scale scientific applications and to quantify the cost of remote replication for write operations, we compare the performance of rNFS against single server configuration with the above Atlas applications. From the evaluation results presented in Section 3.2, we can draw two conclusions. First, we observe that server replication can significantly improve these applications' performance by reducing their first-time access latency to read the input data and by distributing the verification of data access to a nearby server. Second, the penalty of file replication is slight for applications that write their output results at a moderate rate.

Second, to evaluate our design for synchronized access support, we compare the performance of rNFS against single server access with a simple emulated edit-and-run example. The experiment results presented in Section 3.3 suggest that in most cases, the performance penalty to guarantee synchronized access in rNFS is negligible.

3.2. Performance Comparison

To evaluate the presented replication scheme, we compare the performance of Atlas `Digitization` and `Reconstruction` with three different distribution models. In each distribution model, we ran the Atlas applications on a pair of NFSv4 clients. We set the RTT between the two clients to 120 msec, the measured `ping` time from our experimental test bed to CERN. Figure 2 illustrates the experimental setups. As shown, in the Local-Remote distribution, the clients access data from a single NFS server lo-

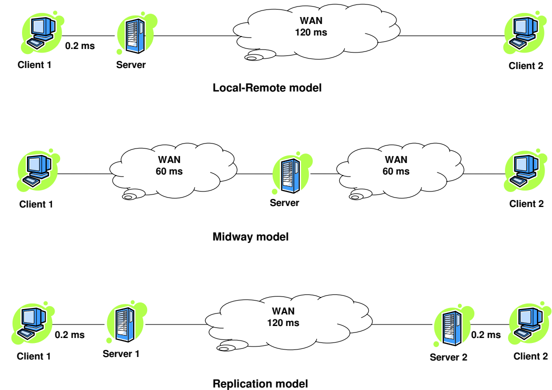

Figure 2: Atlas evaluation experiment setup.

The figure illustrates the experimental setup for the Atlas evaluations. We compare the performance of three distribution models and use two clients in each of them. In Local-Remote model, the clients access data from a single server located on one client's LAN. In Midway model, we place a single server half way between the two clients. In Replication model, we place a replication server on each of the client's LAN.

cated on one client's LAN. In the Midway distribution, we place a single NFS server half way between the two clients. In the Replication distribution model, we place a replication server on each of the client's LAN.

In the experiments presented in this subsection, the number of events to test is set to 100, with each client processing 50 events. Our experiments use the `Digitization` and `Reconstruction` software from the Atlas 10.0.4 installation package. For `Reconstruction`, we applied all the algorithms included in the default installation.

For Local-Remote and Midway distribution models, we present the performance measured with both cold and warm client caches. With Replication, cache temperature does not influence the run time of either application: un-cached items are retrieved from the nearby replication server, and the cost of those retrievals is minuscule in the context of the overall run times of the applications.

Figures 3 and 4 show the run times for the Atlas `Reconstruction` and `Digitization` applications. As the results show, Replication outperforms Local-Remote and Midway for both applications. To see just where the performance improvements come from, we further divide the applications into three phases and measured the time spent on each of them.

In the `Setup` phase, the applications prepare their runtime environments. In the `Initialization` phase, the applications read header files and libraries and link them into executables (We measured the initialization time by setting the number of simulation events to 0). The `Execution` phase processes events; we calculate the `Execution` time by subtracting the time spent on the first

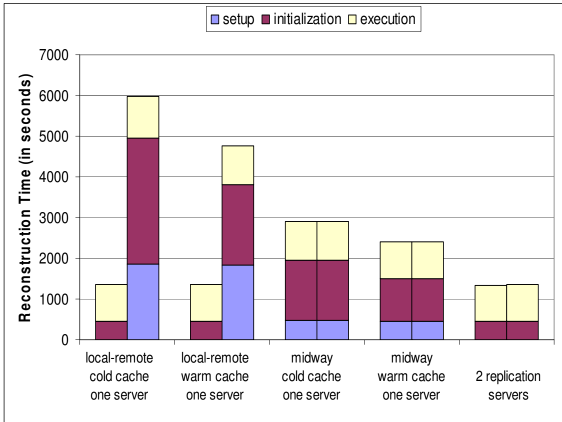

Figure 3: Atlas Reconstruction.
The figure shows the running time for Atlas Reconstruction. In each category, the first column shows the run time measured on the left side client in Figure 2, and the second column shows the measured performance on the other client.

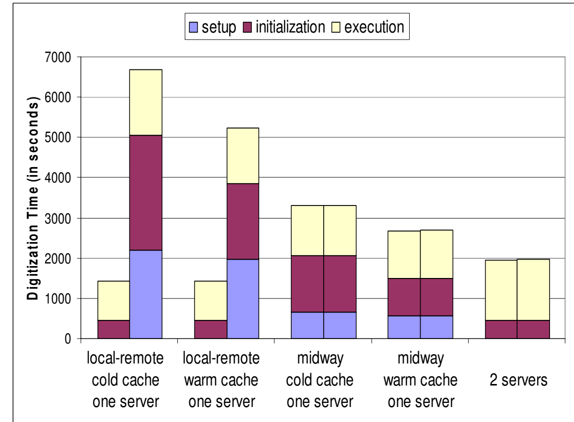

Figure 4: Atlas Digitization.
The figure shows the running time for Atlas Digitization. In each category, the first column shows the run time measured on the left side client in Figure 2, and the second column shows the measured performance on the other client.

two phases from the total running time.

Examining the detailed experimental results presented in Figures 3 and 4, we see that for both applications, most performance benefit of replication comes from `Initialization` and `Setup`. We were surprised to find that even with a warm cache, the performance of these two phases still suffers dramatically as the RTT between the server and the client increases. Taking a close look at that network traffic with a warm cache client, we were surprised to see that a huge number of file open requests are sent during these two phases, as NFSv4 has a delegation scheme that allows a client to perform subsequent open requests locally after the first call in the absence of shared writers. Further examination reveals that most of these open requests were met by "`No entry exists`" error, obviating any potential delegation advantage.

We believe that the applications are issuing these open requests as a way of examining the configuration of the local environment. The cost of doing this on a local or nearby file system is too small to make a substantial difference in the running time, but begins to have an impact as the server is made more and more remote. Here replication helps by allowing the open requests to fail on a nearby server, with the performance comparable to accessing a single local server.

In the `Execution` phase, Atlas `Digitization` produces about 7.2 MB of output data per process. The data is read by `Reconstruction` during the `Execution` phase, whose output size shrinks to 1.4 MB in our measurements. Figures 3 shows that for Atlas `Reconstruction`, the performance of the `Execution` phase is similar in all three distribution models. However, this is not the case in the `Digitization` stage, as observed in Figure 4. There, `Execution` with a remote server is about 50% more costly than `Execution` with a local server, and `Execution` with replication is comparable to the latter.

Although Atlas `Digitization` generates a significant amount of output, we were still surprised that the performance penalty for remote replication is so high. So we examined the trace data collected during the experiment and find that the high performance cost observed during `Execution` is mainly caused by a significant number of fsync system calls. I.e., more than 900 fsync calls are used with 50 event digitization, compared with 60 fsync calls observed with 50 event reconstruction.

To estimate the performance of Atlas `Digitization` without the impact of the aggressive use of synchronous writes, we eliminate these fsync calls and re-run the experiment. Figure 5 shows the re-measured results. As the evaluation data demonstrates, the performance difference between local server access and replication becomes smaller, i.e., `Execution` with replication is about 20% slower than `Execution` with local access. The remained performance overhead is caused by the large bursty writes that exhausts the client's cache.

We have reported this observation to the Atlas developers. It seems that the overwhelming use of fsync is an implementation issue rather than necessaries. However, such kinds of problem may not be rare in practice since most programs in use today are developed in local environments.

Because the four Atlas stages are typically run together as a pipeline, the above problem can be avoided by keeping intermediate outputs in local temporary files and write only final results over distributed file systems. Researchers have shown that a diamond-shaped storage profile is a char-

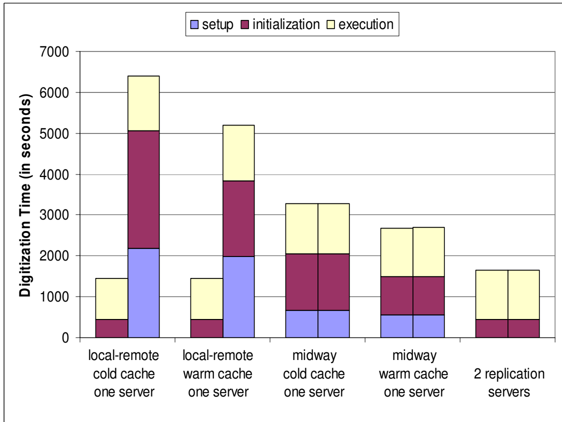

Figure 5: Atlas Digitization without fsync.
The figure shows the running time for Atlas Digitization with fsync calls removed.

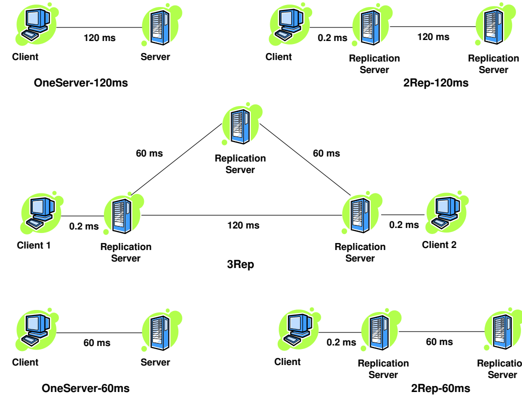

Figure 6: Synchronized access evaluation experiment setup.
The figure illustrates the experiment setup for synchronized access evaluation. We measured the time to overwrite a file or synchronously read a modified file in five distribution models. In OneServer-120ms and OneServer-60ms, the client accesses a single server with the RTTs of 120ms and 60ms, respectively. In 2Rep-120ms and 2Rep-60ms, two replication servers are used with the intermediate RTTs of 120ms and 60ms, respectively. In both distributions, the client connects to a replication server in its LAN. In 3Rep, three replication servers are used with the RTTs among them set to 60ms, 60ms, and 120ms; the writer (client1) and the reader (client2) locate remote from each other, and each connects to a server in its LAN.

acteristic behavior among scientific applications [25]. I.e., small inputs are expanded by early stages into large intermediate results, which are often reduced by later stages to small results. This observation implies that it is more efficient to store intermediate results in local storage rather than distributing them remotely. However, when making such decisions, users should also consider the tradeoff between performance and the cost to re-compute intermediate results if a failure occurs.

We notice that a fundamental problem reflected here is that applications usually use fsync to require different consistency guarantees without distinction. The cost of doing this in a local file system is small, but starts to impact performance as the system becomes widely distributed. As an alternative solution, we suggest that applications use asynchronous fsync to require data durability guarantee, and use fsync and open synchronization flags to coordinate concurrent accesses. We note that distinguishing these different consistency requirements not only allows applications to utilize a consistent mutable replication system at little overhead, but also helps to reduce performance cost in single remote server access.

3.3. Performance of Synchronized Access

This section evaluates the performance of rNFS when synchronized access guarantee is required. In particular, we take a simple edit-and-run example in which a user edits a file on one client and starts running it on another client. For evaluation purpose, we emulate this example by overwriting a file on one client and then immediately reading it from another client. To guarantee read-after-write synchronization, the first client (writer) issues a fsync call after overwriting the file, and the second client (reader) uses synchronous reads.

Below, we first describe the experiment setup. After that, we report the execution time measured on the writer and the reader, respectively.

We measured the time to overwrite a file and/or to synchronously read a modified file in five different distribution models, as illustrated in Figure 6. In OneServer-120ms and OneServer-60ms distributions, the client accesses a single remote server with the RTTs of 120ms and 60ms, respectively. In 2Rep-120ms and 2Rep-60ms distributions, two replication servers are used with the intermediate RTTs of 120ms and 60ms, respectively. In 3Rep distribution, we construct an unbalanced configuration by placing three replication servers with the RTTs among them set to 60ms, 60ms, and 120ms.

Figure 7 shows the total time measured on the writer to overwrite and then fsync a file. In general, replication outperforms single remote server access with the same distribution RTT. The performance benefit comes from the reduced number of remote messages sent to open the file and to check the file's access mode and attributes.

In rNFS, the primary server needs to guarantee that all replication servers have acknowledged its role when it receives a synchronous write request. After receiving replies from all other replication servers, the primary server can reply to a client write request as soon as it gets acknowl-

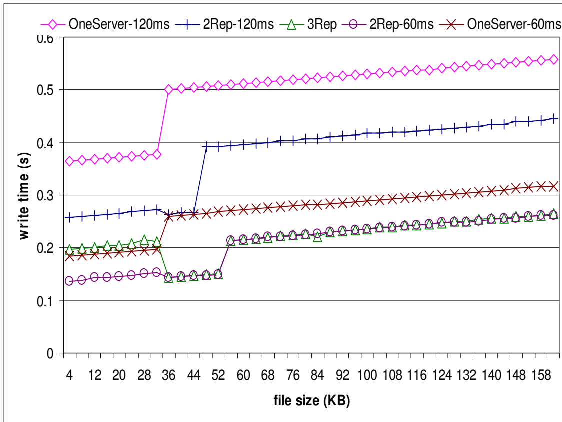

Figure 7: Synchronized writes.
The figure shows the measured time to overwrite & fsync a file as the file size increases. The experiment setup is shown in Figure 6.

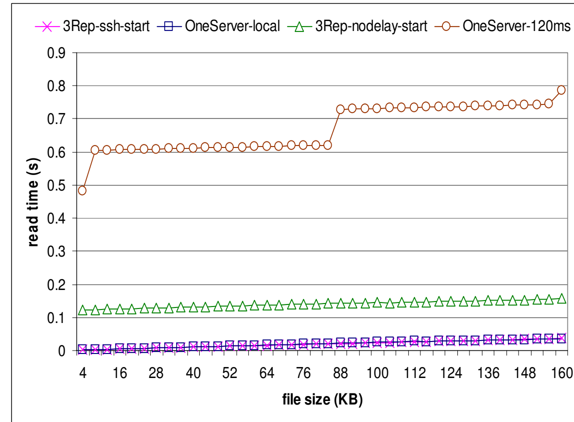

Figure 8: Synchronized reads.
The figure shows the measured time to synchronously read a file that is just modified on another client. In the experiment, 3Rep-ssh-start and 3Rep-nodelay-start both use the 3Rep configuration as depicted in Figure 6. In 3Rep-ssh-start, we start reading the file by sending an ssh command from the writer after writes complete. In 3Rep-nodelay-start, we starts reading the file immediately after file modification, without the cross-wire latency between the two clients. As comparison, we also present the time measured with both the writer and the reader connecting to a single local server and a single remote server, as represented by OneServer-local and OneServer-120ms respectively. In OneServer-120ms, the RTT between the server and the client is set to 120ms.

edgments from half of the other replication servers. So we expect that the synchronous write performance in rNFS is dictated by the majority RTT rather than the longest RTT among the replication servers. The measured experiment results validate our prediction. As observed, the execution time measured in 3Rep distribution is close to that measured in 2Rep-60ms distribution. A slightly longer delay is observed when the file size is small, corresponding to the waiting time when the primary server processes the first synchronous write request. The delay disappears as the file size becomes large, because in those cases, when the first synchronous write request triggered by fsync reaches the primary server, it has already received the acknowledgments from all other replication servers.

After evaluating the write performance during synchronized access, we now move to the reader side.

Figure 8 shows the time to synchronously read a file that is just modified by the writer. To evaluate the performance when read forwarding occurs, we pay special attentions to the 3Rep distribution. In rNFS, the primary server responds to the close request immediately but delays replication re-enabling untill all file updates have been acknowledged by every active replication server. With an unbalanced replication server distribution, a slow or remote server can fall behind from file modification with a burst of writes. So when the reader starts accessing the file, the replication on the server it connects to may still be disabled. As mentioned, the replication server forwards the client synchronous read request to the primary server in this case.

In the experiments, we first start reading the file on the reader by immediately sending a ssh command from the writer after file modification finishes. The network delay between the writer and the reader is set to 120ms, as the experiment setup depicts. However, with this starting method, no forwarded reads are observed because the latency of sending an ssh command dominates the delay of re-enabling replication. As observed, the read performance in such cases is similar to that when both the writer and the reader are connected to a single local server.

For evaluation purpose, we artificially start reading the file right after file modification. In this situation, we observe that the first read request from the reader is forwarded to the primary server. After that, the replication of the file is re-enabled in the system. So the subsequent read requests are processed by the nearby server that the client connects to. As Figure 8 shows, the performance with 3Rep-nodelay-start stays nearly the same as the file size increases, compared with the climbing delay observed when reading the file from a single remote server. The overhead caused by the forwarded read request corresponds to the performance difference observed between 3Rep-nodelay-start and reading a single local server, which is about the same as the RTT between the replication server and the primary server.

In summary, the evaluation results presented in this section show that usually users observe no additional cost during synchronized access in rNFS; a slight performance penalty is charged in the case that read forwarding does occur, but even then, the performance of rNFS still significantly outperforms remote server access.

4. Related Work

In distributed file systems, various consistency guarantees have been introduced. The most stringent guarantee, **strict consistency**, assures that all clients see precisely the same data at all times. Although semantically ideal, strict consistency can be detrimental to performance and availability in networks with high latency, many clients, and the potential for partition. On the other end of the spectrum, consistency guarantees are abandoned altogether, e.g., in P2P systems that strive to maximize availability [20, 18, 22], or are replaced by heuristics for addressing conflicts when they happen [24, 14], i.e., **optimistic replication**. To balance the benefit of replication with the cost of guaranteeing consistent access, some distributed file systems provide **read-only** access to replicated files, side-stepping update consistency problems altogether [23, 4, 26].

We observe that although optimistic replication has been widely studied, few applications in reality are prepared to deal with the conflicts that might happen. Even if applications can provide such support, conflict resolution must be performed carefully; otherwise, the cost to reproduce data, if possible, can be considerable. The lack of consistency guarantees makes it infeasible for scientific collaborations which require reliable and coordinated data access.

For its superior read performance, read-only replication has been favored in the current Grid experimental platform [8]. With read-only replication, once a file is declared as shared by its creator, it cannot be modified. An immutable file has two important properties. I.e., its name may not be reused and its contents may not be altered. While simple, read-only replication has several deficiencies. First, it fails to support complex sharing behavior, e.g., concurrent writes. Second, to guarantee uniqueness of file names, file creation and retrieval require a special API, which hinders using the software developed in the traditional computing environment for global collaborations.

Various middlewares have been developed with the goal to facilitate data access on the Grid. **Storage Resource Broker (SRB)** [5] utilizes metadata catalog service to allow location-transparent access for heterogeneous data sets. **NeST** [6], a user-level local storage software, provides best-effort storage space guarantees, mechanisms for resource and data discovery, user authentication, quality of service and multiple transport protocol support, with the goal to bring appliance technology to the Grid. The **Chimera** system [10] provides a virtual data catalog that can be used by applications to describe a set of programs, and then track all the data files produced by executing them. The work is motivated by observing that a lot of scientific data is derived from other data by the application of computational procedure, which implies the need for a flexible data sharing and access system.

A common missing feature among these middlewares is the lack of supporting fine-grained data sharing semantics. Furthermore, most of these systems provide extended features by defining their own API. In order to use them, an application has to be re-linked with their libraries.

The emerging large-scale scientific collaborations have stimulated the growing research in scientific workload studies. Here we only summarize two recent works that are directly related to our study.

Thain et al. study the workload characteristics of six scientific applications [25] whose workloads are composed of several pipelines. The studied workloads demonstrate three common behaviors: First, small initial inputs are usually expanded by early stages into large intermediate results, which are often reduced by later stages to small results. Second, although users tend to identify large data collections needed by an application, in a given execution, applications usually selects a small working set. And third, significant data sharing are observed for users often submit large numbers of very similar jobs that access similar working sets.

Holtman et al. investigate the data processing requirements that CMS experiments demands and the expected workload characteristics after the LHC collider starts running [13]. Regarding to file access, they point out that the CMS workloads will be dominated by reading, and the stepwise refinement of algorithms will lead to a workload where series of jobs are run over the same input data, with each job containing the refined code or parameter. Sometimes, CMS applications need to randomly access data from data sets that are too large to stage to every machine in a site. Such use cases require accessing files on a large file system local to the site. Furthermore, CMS expects to access files through regular POSIX I/O calls without re-linking with special libraries.

5. Conclusion

Supporting consistent mutable replication in large-scale distributed file systems is traditionally considered too expensive to utilize. The work presented in this paper demonstrates that it is feasible and practical to provide such support with negligible impact on common case performance. In the paper, we describe a replicated file system designed to meet the needs of global collaborations. The system supports a global name space and location independent naming, which facilitates data sharing, distribution, and management. It uses a replication protocol that supports mutable replication with stringent yet flexible consistency guarantees. The evaluation results using a real scientific application show that the presented replication system can significantly improve application performance by allowing them to access data from a nearby server. Furthermore, the performance overhead to support mutable replication and to guarantee consistency is small as long as applications use synchronous write at a moderate rate.

References

[1] ATLAS. `http://atlasinfo.cern.ch/Atlas/Welcome.html`.

[2] LHC. `http://atlasinfo.cern.ch/Atlas/Welcome.html`.

[3] *UNIX man pages: open(2)*, second edition, 1997.

[4] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, and I. Foster. Secure, efficient data transport and replica management for high-performance data-intensive computing. In *Proc. of the Eighteenth IEEE Symposium on Mass Storage Systems and Technologies*, page 13, Washington, DC, USA, 2001. IEEE Computer Society.

[5] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proc. of CASCON'98*, 1998.

[6] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Flexibility, manageability, and performance in a grid storage appliance. In *Proc. of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC-11)*, July 2002.

[7] M. Carson and D. Santay. NIST Net: a Linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, 2003.

[8] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*.

[9] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[10] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proc. of the 14th Conference on Scientific and Statistical Database Management*, 2002.

[11] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[12] A. Hisgen, A. Birrell, T. Mann, M. Schroeder, and G. Swart. Availability and consistency trade-offs in the Echo distributed file system. In *Proc. 2nd IEEE Workshop on Workstation Operating Syst*, 1989.

[13] K. Holtman. Cms data grid system overview and requirements. The Compact Muon Solenoid (CMS) Experiment Note 2001/037, CERN, Switzerland, 2001.

[14] P. Kumar and M. Satyanarayanan. Supporting application-specific resolution in an optimistically replicated file system. In *Workshop on Workstation Operating Systems*, pages 66–70, 1993.

[15] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proc. of 13th ACM Symposium on Operating Systems Principles*, pages 226–38, 1991.

[16] P. Mockapetris. Domain names - concepts and facilities. STD 13, RFC 1034, November 1987.

[17] P. Mockapetris. Domain names - implementation and specification. RFC 1035, November 1987.

[18] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of 5th Symposium on Operating Systems Design and Implementation*, 2002.

[19] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. In *Proc. of the 2nd International System Administration and Networking Conference (SANE2000)*, page 94, 2000.

[20] G. J. Popek, R. G. Guy, T. W. Page, Jr., and J. S. Heidemann. Replication in Ficus distributed file systems. In *IEEE Computer Society Technical Committee on Operating Systems and Application Environments Newsletter*, volume 4, pages 24–29. IEEE Computer Society, 1990.

[21] G. Project. Gridftp: Universal data transfer for the grid, 2000. white paper.

[22] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proc. of 5th Symposium on Operating Systems Design and Implementation*, 2002.

[23] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West. The ITC distributed file system: principles and design. In *Proc. of the 10th Symposium on Operating Systems Principles*, pages 35–50, 1985.

[24] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[25] D. Thain, J. Bent, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Pipeline and batch sharing in grid workloads. In *Proc. of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 152, Washington, DC, USA, 2003. IEEE Computer Society.

[26] B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw. Legionfs: a secure and scalable file system supporting cross-domain high-performance applications. In *Proc. of the 2001 ACM/IEEE conference on Supercomputing*, pages 59–59, New York, NY, USA, 2001. ACM Press.

[27] J. Zhang and P. Honeyman. Consistent file replication for wide area collaboration. Technical Report CITI-TR-05-4, Ann Arbor, MI, USA, July 2005.